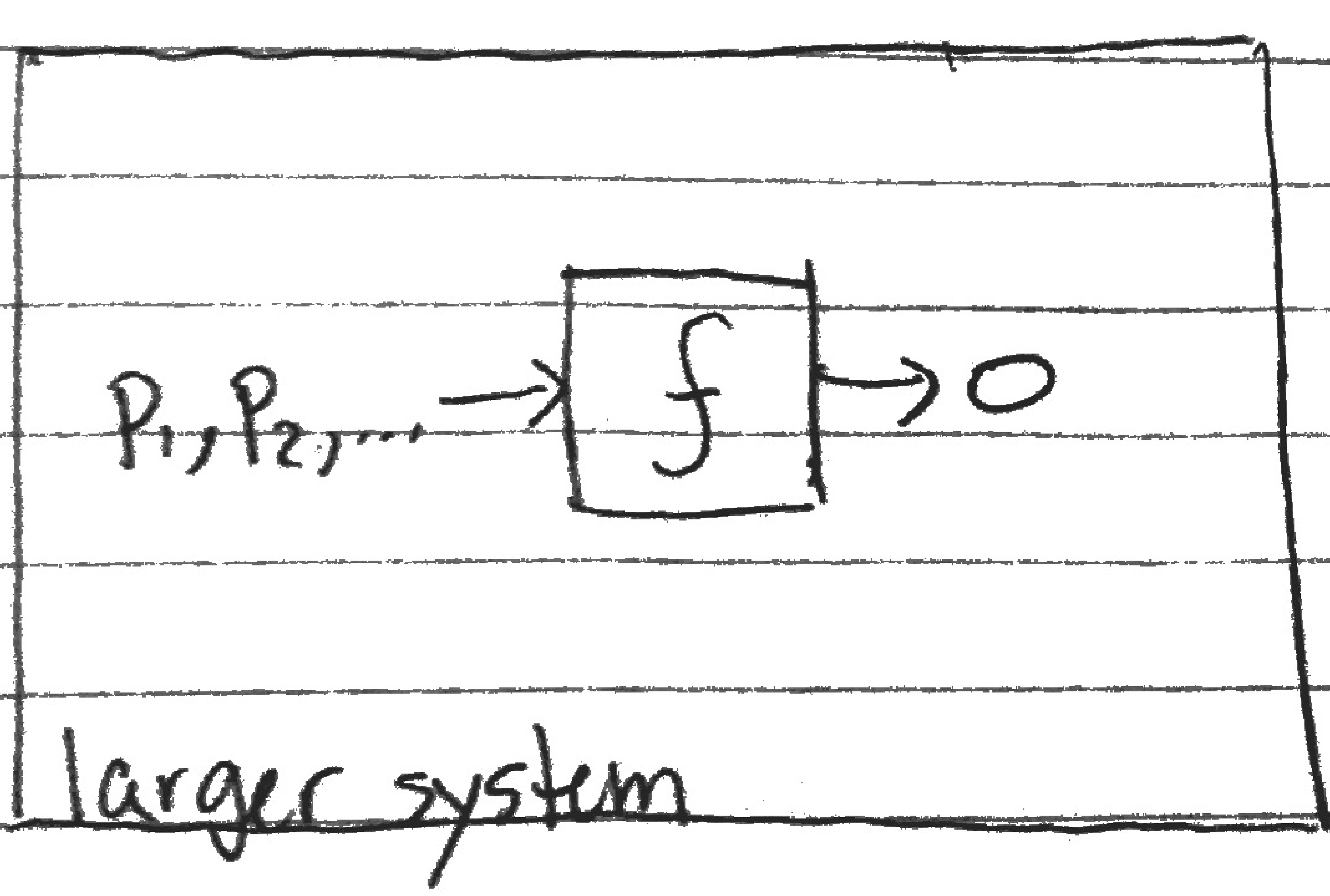
What is a pure function?

defn: A function f with input type A and output type B ($A \Rightarrow B$) is a computation that relates every value of $a \in A$ to exactly one value $b \in B$ such that b is determined solely by the value of a.

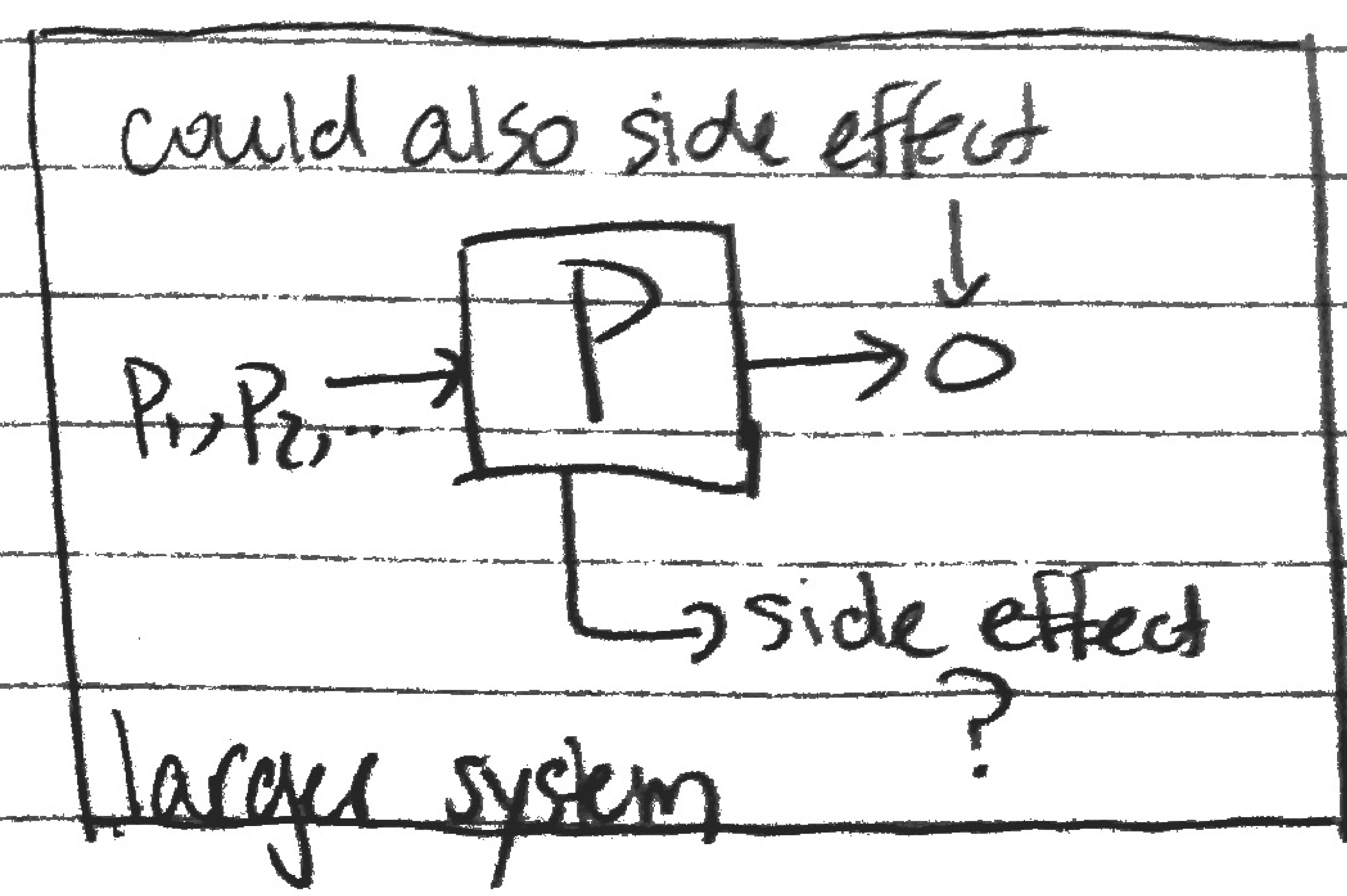
$\downarrow$

A function has no observable effect on the execution of a program other than to compute a result given its inputs.

(Pure) Function

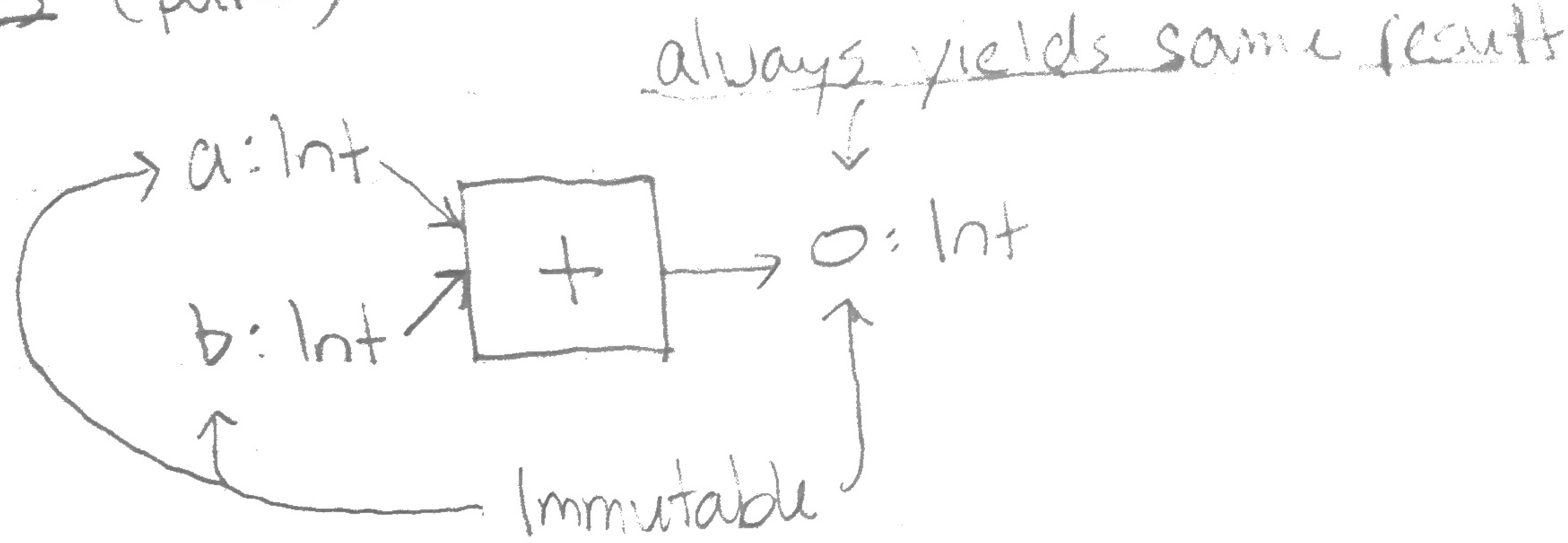


Procedure

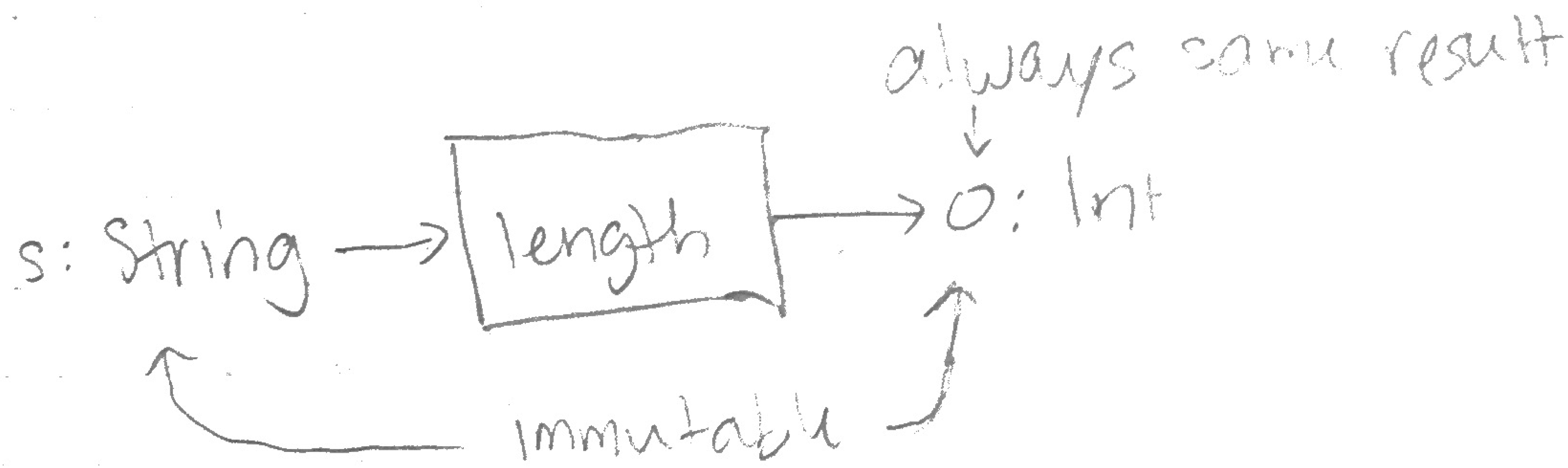


Examples (pure)

1.

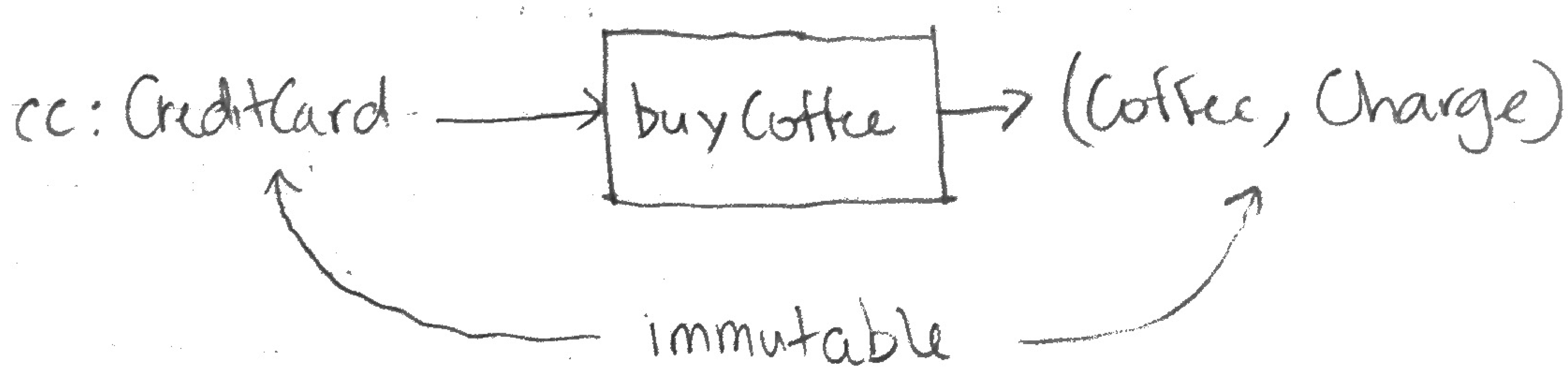


2.



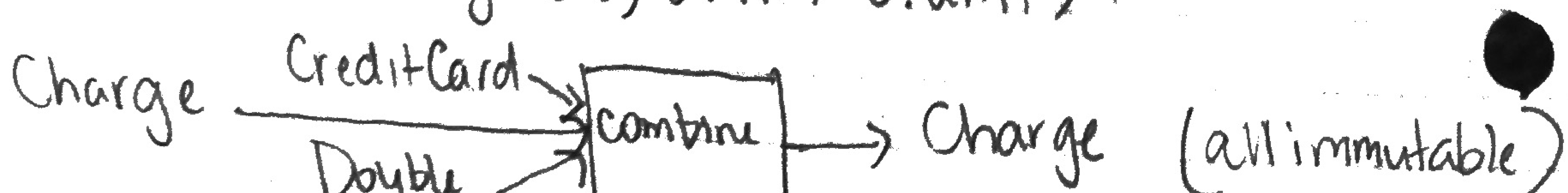
3.

```
def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
  val cup = new Coffee
  (cup, Charge(cc, cup.price))
}
```



4.

```
class Charge(cc: CreditCard, amt: Double) {
  def combine(o: Charge): Charge =
    if (cc == o.cc)
      Charge(cc, amt + o.amt)
```



Because these are pure functions, they can be easily used anywhere with no impact to the rest of the system — modular + composable

↳ this is known as referential transparency (RT).

(A property of expressions in general not just functions.)

(An expression is any part of your program that can be evaluated to a result)

eval(exp) → result

← expression

s> 2 + 3

↳ 5: Int ← result

The evaluation of this expression results in the same value 5 every time!

eval(2+3)
eval(eval(2) + eval(3))
eval(2 + eval(3))
eval(2 + 3)
eval(5)
5

[This is what it means to be referentially transparent!]

↳ (in fact, if we saw 2+3 in a program we could replace it with 5 and it wouldn't change the meaning of our program)

Referential Transparency + Purity

<u>defn</u>: An expression e is RT if, for all programs p, all occurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p.

A function f is <u>pure</u> if the expression f(x) is RT ∀ RT x.

RT → everything a function does is represented by the value it returns

This enables a natural mode of reasoning about program evaluation called the <u>substitution model</u>

(computation proceeds like we'd solve an algebraic equation)

<u>Example</u>: Boolean Expressions

(1) !true → false
(2) !false → true
(3) true && e → e
(4) false && e → false
(5) true || e → true
(6) false || e → e

short-circuit evaluation

((T && !T) || (!F || T)) && ((T || F) || !F)

Is this T or F?
Which rules are applied?

How do we add if/else to our rules?

(7) if (true) e_1 else e_2 $\rightarrow$ e_1
(8) if (false) e_1 else e_2 $\rightarrow$ e_2

Can you define a rule for && and || without using && and ||?

$$\boxed{\begin{array}{l} e_1 \;\&\&\; e_2 \rightarrow \text{if } (!e_1) \text{ false else } e_2 \\ e_1 \;||\; e_2 \rightarrow \text{if } (e_1) \text{ true else } e_2 \end{array}}$$

How about XOR? e_1 ^ e_2 is true when e_1 != e_2

$$\boxed{e_1 \wedge e_2 \rightarrow (e_1 \;\&\&\; !e_2) \;||\; (!e_1 \;\&\&\; e_2)}$$

you can use && and ||

How does this relate to Scala code?

```
s> val x = "Hello, World"
s> val r1 = x.reverse
s> val r2 = x.reverse
```

r_1 and r_2 are the same

↳ what if we substitute x with expr referenced by x?

```
s> val r1 = "Hello, world".reverse
s> val r2 = "Hello, world".reverse
```

$r_1 == r_2$ is true

↳ x is RT

How about this?

```
s> val x = new StringBuilder("Hello")
s> val y = x.append(", world")
s> val r1 = y.toString
s> val r2 = y.toString
```

r_1 and r_2 are the same

* StringBuilder.append is a side effecting function.

```
s> val x = new StringBuilder("Hello")
s> val r1 = x.append(", World").toString
s> val r2 = x.append(", World").toString
```

r_1 and r_2, not the same!

StringBuilder.append is not pure

point: hard to reason about side effecting code because you need to understand the larger context.

Higher-order functions

New idea: functions are values

↳ just like Int, String, List, ...

Functions can be assigned to variables, stored in data structures, and passed as arguments to functions

* when writing functional programs – it is useful to write functions that accept other functions as arguments *

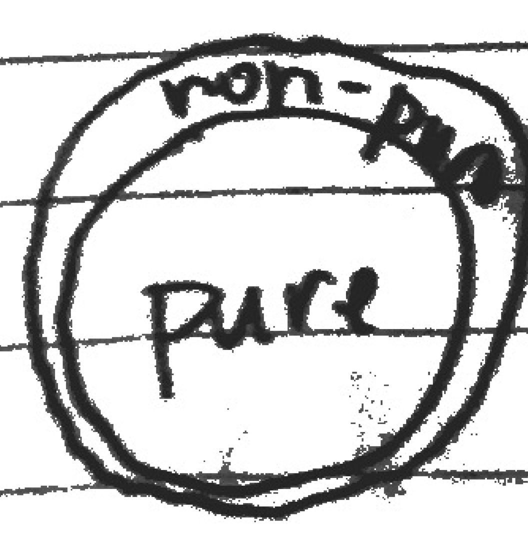
Example: higher-order / Absolute01.scala

(Now, how might we adapt this program to print both absolute value and factorial?)

Example: higher-order / Higher02.scala

Example: higher-order / Higher03.scala

Functional programs are typically written with a rich functional/pure core and a thin non-pure wrapper.



Polymorphic Functions

↳ **monomorphic** – functions that operate over single type.

polymorphic – functions that operate over **many** types.

↕ we often recognize that we can make a function more **generic** (more reusable) if they can operate over many types of data.

↓

rather than writing a function for each type.

Example: def findFirst(ss: Array[String], key: String): Int

monomorphic

How might we "find first" over an array of type T?

↓ polymorphic

def findFirst[A](as: Array[A], p: A => Boolean): Int

Example: higher-order/Poly04.scala

Although, you could define function by name to pass to higher-order polymorphic functions,

it is often more more convenient to use <u>anonymous functions</u>.

```
s> (x: Int) = x == 3
   ↳ (Int) => Boolean

s> (x: Int, y: Int) => x + y
   ↳ (Int, Int) => Int
```

<u>Example</u>: higher-order/Polyos.scala

Implement the following function:

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C
```

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =
    (b: B) => f(a, b)
```